SQUID で Julia プログラミング

宮武 勇登
大阪大学 サイバーメディアセンター

1. はじめに

Julia（図 1）は、2009 年頃から開発が始まり 2012 年に公開された比較的新しい汎用プログラミング言語である。近年、人工知能やデータサイエンス分野で注目を集めており、データサイエンスの次世代言語とも言われている。

図 1：Julia のロゴ

まず、Julia の際立った特徴を幾つか挙げてみよう。

- Julia は Python, Matlab, R などと同様に動的型付けの言語である。プログラムの構造は、C 言語や Fortran などよりも Python, Matlab, R などに近く、簡潔で高水準なコードを書くことができる。したがって、学習コストは比較的小さく、生産性は高い（配列のインデックスは 1 スタートであり、科学技術計算の基盤である線形代数との親和性も高い(1)）。
- 一般に、動的型付けの言語は、生産性と実行効率の間にトレードオフがあることが多く、生産性が優れている反面、C 等と同等の実行速度を期待することは難しい。しかし、Julia は細かい調整をしなくとも、C 等に匹敵する実行速度を期待できることが多い。ベンチマーク問題に対する様々な言語の実行速度の比較が(2)で公開されている。動的型付けの言語である Julia では、基本的に型の情報を付与する必要はなく、実行時のコンパイルにおいて十分に最適化されたコードになることも多いが、型情報を付与することでさらなる高速化が達成されることもある。
- Julia で実現することが直ちには難しいレベルのチューニングが施された他言語のプログラムも、比較的容易に利用することができる（例えば C で書かれたライブラリを Julia から呼び出せる）。

2012 年に公開されてから、しばらくバージョンは 0 台であったが、2018 年に v1.0.0 が公開されてからは、以前のプログラムが動かなくなるようなレベルの言語仕様の変更はほぼなく、利便性も向上している。Python 等と比較すると、日本語の解説書（例えば(3)）は限られてはいるものの（とはいえ、ブログなどを通した情報発信は盛んに行われているし、基本的に英語ではあるが公式 web ページから十分な情報を入手できる）、以上のような特性から、Julia の利用者も年々増加しており、当然、スーパーコンピュータ上で Julia を利用する要望も高まっている。

大阪大学サイバーメディアセンターで 2021 年 5 月より運用が開始されたスーパーコンピュータ SQUID では、Julia を利用することができる（Python や R も利用可能）。本稿執筆時点で、Julia に関する情報は十分に公開されているとは言えないが、SQUID 上で Julia を利用できるという事実を広く普及するためにも、本稿では、日頃ラップトップ PC やワークステーションで Julia を利用しているユーザ（特にスーパーコンピュータを使った並列計算の経験があまり多くはないユーザ）を念頭に、SQUID 上での Julia の利用についてその基本を概観する。

2. Julia の実行方法

一般論として、通常の Julia の利用方法は Python と似通っている。まず、気軽に利用できるのは対話型環境 (REPL) であろう。また、Julia のプログラム（例えば sample.jl）があれば、コマンドラインに

$ julia sample.jl

と入力すればプログラムを直接実行できる。他にも、Jupyter Notebook（とその次世代版である JupyterLab）も便利でよく利用されている。これは、ウェブブラウザ上で動作する対話型実行環境である。

SQUID 上でも、通常通りの利用方法が可能だが、注意点も少なくない。

まず、SQUID には HPC フロントエンドと HPDA フロントエンドの 2 つのフロントエンドがある。REPL を利用したりコマンドラインから直接プログラムを実行したりする場合は、前者の HPC フロントエンドを利用し、Jupyter Notebook を利用する場合は、後者の HPDA フロントエンドを利用することとなっている（本稿執筆時点で JupyterLab はサポートされていない）。個人の PC などでの通常の利用では、REPL を起動した上で

julia> using IJulia

julia> notebook()

として Jupyter Notebook を利用することも多いが、SQUID 上ではそのような使い方は想定されていない。言い換えれば、HPC フロントエンドで REPL を起動し、そこから IJulia パッケージを利用して Jupyter Notebook を起動することはできず、万が一このような操作を行ってしまうと何らかの不具合が生じうるため注意が必要である。

2.1 HPC フロントエンド

利用方法はいたってシンプルであり、HPC フロントエンドにログイン後、コマンドラインに

$ module load BaseJulia/2021

と入力すれば、直ちに利用できる。実際、コマンドラインに

$ julia

と入力すれば、次のような REPL が起動する。

```
[      @squidhpc3 ~]$ module load BaseJulia/2021
Loading BaseJulia/2021
  Loading requirement: julia/1.6.1
[      @squidhpc3 ~]$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.6.1 (2021-04-23)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

また、REPL ではなくコマンドラインから Julia ファイルを実行するには、

$ julia sample.jl

のように入力すればよい。

ただし、原理的にはこのように利用できるものの、フロントエンドノードは多数のユーザで共有していることから、簡易的な動作確認のみを想定しており、そうではないプログラムの実行を行ってはならない。

したがって、実際にプログラムを実行するためには、ジョブスクリプトにて計算ノードへジョブを投入し、バッチ利用でプログラムを実行する必要がある。以下はジョブスクリプトの一例である。

```
#!/bin/bash
#PBS -q SQUID
#PBS --group=【グループ名】
#PBS -l elapstim_req=1:00:00
module load BaseCPU/2021
module load BaseJulia/2021
cd $PBS_O_WORKDIR
julia sample.jl
```

#PBS から始まる行は、利用する計算機のリソースや環境の指定となっている。使用する CPU コア数の要求値等を指定することもできる。このような計算機のリソースや環境については Julia 特有のことはなく、マニュアルを参考に記述すればよい。Julia 特有のことは、モジュール BaseJulia/2021 を読み込むことと、最終行が julia の実行コマンドになることの二点である。

なお、Julia ではパッケージを利用した計算を行うことがほとんどであり、基本的には事前にフロントエンドノードの REPL を使って追加を行うか、あるいは、プログラムの中にパッケージを追加する指示を含めておく必要がある。

2.2 HPDA フロントエンド

Jupyter Notebook を利用するには、HPDA フロントエンドにログインする必要がある。ログイン後、Julia カーネルが利用可能な jupyter コンテナを取得し（コンテナの取得は一度でよい）、コンテナを起動すれば Jupyter Notebook を利用できる。Python, R, Julia のそれぞれについてカーネル環境が予め構築されたコン

テナが用意されているため、Python や R を利用する際には別のコンテナを利用する必要がある。また、複数のコンテナを同時に起動できないことにも注意が必要である（別ノードで同種のコンテナを重複起動したい場合には、sif ファイルのパスを変更することで可能となる）。コンテナの取得・起動・停止のコマンドは以下の通りである。

【コンテナの取得】

$ singularity pull jupyter-julia.sif oras://cntm:5000/master_image/jupyter-julia:1.0

【コンテナの起動】

$ run_jupyter_container.sh -k julia

【コンテナの停止】

$ stop_jupyter_container.sh -k julia

Jupyter Notebook を起動する際には、パスワード認証があり、続いてコンテナ起動時にターミナルに表示される jupyter トークンを入力する。jupyter トークンによる認証も成功すると、以下のような Jupyter Notebook の初期画面が表示される。

コンテナ起動時に jupyter_notebook というディレクトリが作成され、Jupyter Notebook からは jupyter_notebook 内に追加あるいは作成したディレクトリやファイルを利用できる。一旦 Jupyter Notebook を起動すれば、その後の利用方法はローカルな環境で利用する場合と変わらない。

このように、SQUID 上でも Jupyter Notebook を利用できるのは大変便利ではあるが、フロントエンドでしか利用することができず、やはり動作確認程度（例えば、図 2 のような簡単な図の描画の確認等）の利用しか想定されていない。

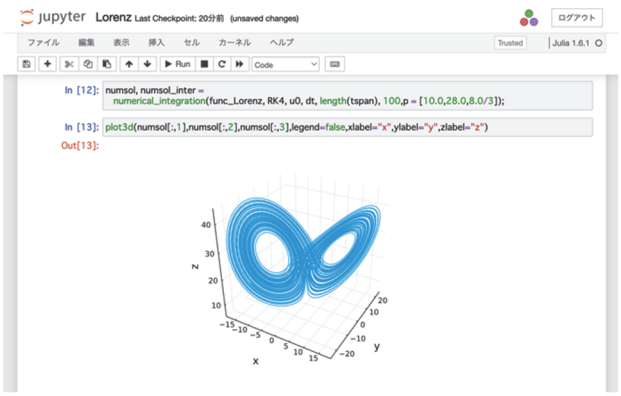

図 2：Jupyter Notebook の作業の一例

3. Julia の並列計算

Julia では、実装の容易な simd 計算等に加えて、4 種類の並列計算がサポートされている。

・非同期のタスク

厳密には並列計算ではないが、複数のスレッドでタスクをスケジュールできる。

・マルチスレッド

メモリを共有しながら、複数のスレッドまたは CPU コアで同時にタスクをスケジュールできる。これは、PC または単一の大規模マルチコアサーバーで並列処理を行う最も簡単な方法である。

・分散コンピューティング

別々のメモリ空間で複数の Julia プロセスを実行できる。

・GPU コンピューティング

Julia GPU コンパイラは、GPU 上で Julia コードをネイティブに実行する機能を提供する。

並列計算の基礎的事項については公式 web ページ(4)にまとめられている。

4. 1 ノード内の並列計算の一例

1 ノードを使用した場合の並列計算の簡単な例を紹介する。以下はコイン投げをして、表の出た回数をカウントする 2 つの関数の比較である。

```
using Distributed
Distributed.addprocs()
@everywhere using BenchmarkTools
@everywhere using ProgressMeter
```

```
function count_heads()
    c::Int = 0
    for i = 1:200000000
        c += Int(rand(Bool))
    end
    return c
end

@everywhere function count_heads_dist()
  nheads = @distributed (+) for i = 1:200000000
      Int(rand(Bool))
  end
  return nheads
end

@benchmark count_heads()
@benchmark count_heads_dist()
```

まず、Distributed.addprocs()によりワーカプロセスが追加されるが、通常はコア数分追加される。SQUIDの場合、1ノードあたり38コアのプロセッサが2つ搭載されているため、76個のワーカプロセスが追加される。2つの関数のうち、前者が1コアのみを用いた通常の計算に対応し、後者が複数のコアを利用した計算に対応している。実際にBenchmarkToolsパッケージを使って計算時間を測定し、計算時間の平均を表示すると

- count_heads()
  → 573.124 s
- count_heads_dist()
  → 8.699 s

となり、おおよそ66倍の高速化が達成されていることが分かる。また、ジョブスクリプトにおいて、julia -p 38 sample.jl のようにプロセス数を指定することもできる（その場合、プログラム中のDistributed.addprocs() の一行は削除しておく）。このとき1プロセッサ（38コア）までの指定であれば、おおよそ指定した数から期待する高速化が達成される。

5. おわりに

本稿では、近年注目が高まっている Julia 言語について、特に SQUID 上での利用方法について紹介した。おそらく、スーパーコンピュータ上で Julia を本格的に利用しているユーザは他の言語と比較して少数であると思われることから、本稿では、比較的プログラミングが容易である 1 ノード内の並列計算の例を紹介した（もちろん、3 節で述べた内容を含むより高度な計算も可能である）。このような並列計算であれば、複数のコアを有するコンピュータであれば、ラップトップ PC からスーパーコンピュータまで基本的に同一のプログラムで実行可能である。したがって、手元のコンピュータを使ってテストを行えば、だいたいのイメージを掴むことが可能である。

本稿が、Julia ユーザが今後スーパーコンピュータを利用して大規模計算を行うことの一つのきっかけとなれば幸甚である。

参考文献

(1) E. Darve, M. Wootters, Numerical Linear Algebra with Julia, SIAM, (2021).

(2) Julia Micro-Benchmarks, https://julialang.org/benchmarks/

(3) 進藤裕之、佐藤健太、1 から始める Julia プログラミング、コロナ社、(2020).

(4) https://docs.julialang.org/en/v1/manual/parallel-computing